1 GNU Radio

GNU Radio[1] is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in hobbyist, academic and commercial environments to support both wireless communications research and real-world radio systems.

A software radio is a radio system which performs the required signal processing in software instead of using dedicated integrated circuits in hardware. The benefit is that since software can be easily replaced in the radio system, the same hardware can be used to create many kinds of radios for many different transmission standards; thus, one software radio can used for a variety of applications.

GNU Radio performs all the signal processing. You can use it to write applications to receive data out of digital streams or to push data into digital streams, which is then transmitted using hardware. GNU Radio has filters, channel codes, synchronization elements, equalizers, demodulators, vocoders, decoders, and many other elements (in the GNU Radio jargon, we call these elements blocks) which are typically found in radio systems. More importantly, it includes a method of connecting these blocks and then manages how data is passed from one block to another. Extending GNU Radio is also quite easy; if you find a specific block that is missing, you can quickly create and add it.

Since GNU Radio is software, it can only handle digital data. Usually, complex baseband samples are the input data type for receivers and the output data type for transmitters. Analog hardware is then used to shift the signal to the desired center frequency. That requirement aside, any data type can be passed from one block to another - be it bits, bytes, vectors, bursts or more complex data types.

GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions, where available. GNU Radio Companion(GRC) is a Simulink-like graphical tool to design signal processing flow-graphs.

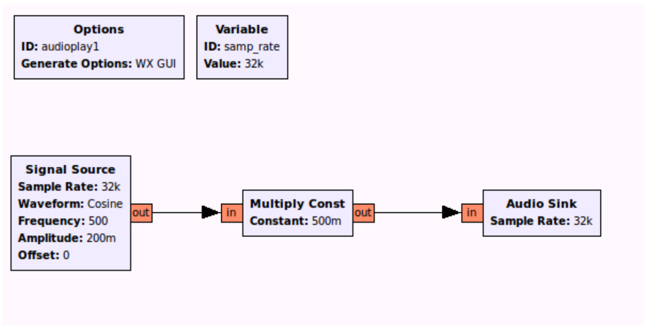
GNU Radio supports several radio front-ends, either natively or through additional out-of-tree modules. We will be using Ettus Research USRP platform and RTL-SDR TV tuners.

1.1 First flow-graph

Flow graphs are graphs (as in graph theory) through which data flows. Many GNU Radio applications contain nothing other than a flow-graph. The nodes of such a graph are called blocks, and the data flows along the edges. The blocks are connected at ports. Data flows into or out of a block through these ports.

Any actual signal processing is done in the blocks. Ideally, every block does exactly one job - this way GNU Radio stays modular and flexible. Blocks are usually written in C++ (might also be Python); writing new blocks is not very difficult.

In order to illuminate this diffuse topic a little, let's start with an example. Our aim is to generate a sinusoid and play it on audio output. Open GNU Radio companion by executing gnuradio-companion on a terminal. Construct a flow-graph as shown in Figure 1.


Options
ID: audioplay1
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 32k

Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 500
Amplitude: 200m
Offset: 0

Multiply Const
Constant: 500m

Audio Sink
Sample Rate: 32k

Figure 1: Playing 500Hz sinusoid on audio output. Take signal source and audio sink block from the list of blocks. Set the data type of signal source to float. Set the frequency to 500Hz and sampling rate to 32k. Set the sampling rate in the audio sink block to 32k. Connect these blocks and execute the flow-graph.

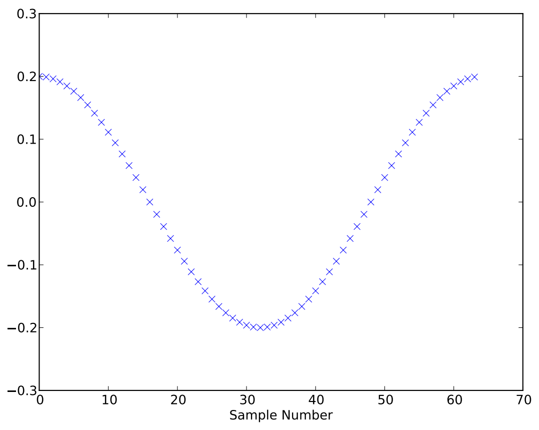


Figure 2: The first 64 samples of output from the signal source.

The first block has no input port. Such a block, with only output ports, is called a source. In an analog fashion, the final block, with no outputs, is called a sink.

So what's happening here? The signal source produces samples of 500Hz cosine sampled at 32kHz per second. The first 64 samples (which corresponds to a complete cycle of cosine in this case) is plotted in Figure 2. These samples are consumed by the multiply constant block, which multiplies each of the input with 0.5. Finally, the samples are passed to the audio output block which plays the incoming samples using computers audio card.

In general, we call whatever a block outputs/consumes, an item. In Figure 1, one item was a float value representing one sample produced by the signal source. However, an item can be anything that can be represented digitally. The most common types of samples are real samples (as before), complex samples (the most common type in software defined radio), integer types, and vectors of these scalar types.

In Figure 1, apart from playing the samples on audio card, everything else is done in software. Since the audio card can only take specific sampling rates(32kHz, 44.1kHz, etc) the rate at which the samples are processed through this flow-graph will be equal to the sampling rate of the audio card.

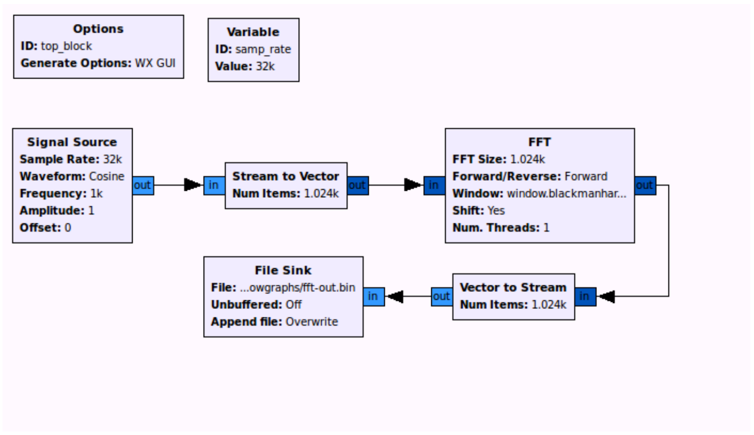
As mentioned before GNU Radio applications are written in python programming language. When we run any flow-graph in GNU Radio companion, it is a python program that runs behind the scene. The python program corresponding to the flow-graph in Figure 1 is given in Listing 1. This program is auto-generated when we run the flow-graph in Figure 1 in grc.

Let us consider another example of computing the FFT of a signal source. The flow-graph is given in Figure 3. In this example, the second block (stream to vector) produces one item(which is a vector of 1024 complex number) for every 1024 input items. So, the rate at which it produces items is 1024 times smaller then the rate at which it consumes items (the fact that it actually produces bytes at the same rate it consumes them is irrelevant here). Such a block is called a decimator, well, because it decimates the item rate. A block which outputs more items than it receives is called an interpolator. If it produces and consumes at the same rate, it's a sync block.

Let us ask the question what is the base sampling rate in Figure 3? Or how many complex numbers produced by the signal source, will be processed through this flow-graph in a second? Well, we can't really answer this question. In this flow-graph all that we are do, is to generate some numbers, do some math on the generated numbers, and save the output to a file. As long as there is no hardware clock present which fixes the rate, sampling rate is meaningless–only relative rates (i.e. input to output rates are important. The computer may handle the samples as fast as it wants. Note that this can cause the computer to lock up by allocating 100% of CPU cycles to your signal processing.

In order to avoid the computer getting busy doing all the math, it is recommended to use a throttle block when we run GNU Radio without any other hardware. The throttle block is tied to computer's hardware clock, and it throttles the number of items passing through it.

Also when we have hardware involving different sampling rates, we might have to use some up-sampling/down-sampling so that we meet the sampling rate requirement of all the hardwares.



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 32k

Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0

Stream to Vector
Num Items: 1.024k

FFT
FFT Size: 1.024k
Forward/Reverse: Forward
Window: window.blackmanhar...
Shift: Yes
Num. Threads: 1

Vector to Stream
Num Items: 1.024k

File Sink
File: ...owgraphs/fft-out.bin
Unbuffered: Off
Append file: Overwrite

Figure 3: Computing FFT

1.2 Complex data type in GNU Radio

Most of the signal processing in GNU Radio is done on complex numbers. In GNU Radio a complex number occupies 8 bytes in memory. It actually consists of two 4-byte floats representing the real and imaginary components respectively.

2 IQ Modulation[4]

In RF communication we transmit our information in a small passband around a high frequency carrier. All the information carried in a real-valued passband signal is contained in a corresponding complex-valued baseband signal. This baseband signal is called the complex baseband representation, or complex envelope, of the passband signal. Any passband signal $s_{\mathrm{p}}(t)$ can be written as

$$s_{\mathrm{p}}(t) = \sqrt{2}s_{\mathrm{i}}(t)\cos(2\pi f_c t) - \sqrt{2}s_{\mathrm{q}}(t)\sin(2\pi f_c t) \tag{1}$$

where,

$$s(t) = s_{\mathrm{i}}(t) + \mathrm{j}s_{\mathrm{q}}(t) \tag{2}$$

is called the complex envelope of $s_{\mathrm{p}}(t)$. In GNU Radio we work with this complex envelope. There are many hardwares available which can be interfaced with GNU Radio to transmit the output from a flow-graph around any carrier that we may choose. Figure 4 compares the spectrum of complex envelope and passband signals. Figure 5 shows the process of IQ modulation. Figure 6 shows the process of IQ demodulation.

3 IQ Modulator Board

We have developed an IQ modulator board at Wadhwani Electronics Labs. This IQ modulator board features Linear Technology's LTC5598 modulator LTC6946-1 synthesizer and PIC18F4550 micro-controller. With the combination of these chips the IQ modulator can transmit in the range 373MHz - 1600MHz. The block diagram of this board is given in Figure 7.

LTC 5598[2] is a high linearity direct quadrature modulator, which works in the range from 5MHz to 1600MHz. It allows direct modulation of an RF signal using differential I and Q signals. The internal block digram of LTC5598 is given in Figure 8. The baseband input $s_{\mathrm{i}}(t)$ is fed through the differential pair BBPI(pin 21) and BBMI(pin 22). The baseband input $s_{\mathrm{q}}(t)$ is fed through the differential pair BBPQ(pin 10) and BBMQ(pin 9). The I/Q baseband inputs consist of voltage-to-current converters that in turn drive double-balanced mixers. The outputs of these mixers are summed and applied to a buffer, which converts the differential mixer signals to a 50Ω single-ended buffered RF output. The four balanced I and Q baseband input ports are intended for DC coupling from a source with a common-mode voltage level of about 0.5V. We will be generating our baseband inputs $s_{\mathrm{i}}(t)$ and $s_{\mathrm{q}}(t)$ from a two channel arbitrary function generator. Since these signals are single ended, we will be feeding BBPI with $s_{\mathrm{i}}(t)$ and BBPQ wit $s_{\mathrm{q}}(t)$. A DC offset(0.5 typical) should be added with $s_{\mathrm{i}}(t)$ and $s_{\mathrm{q}}(t)$ at the AFG to meet common mode voltage requirement. Their differential pair BBMI and BBMQ are tied to

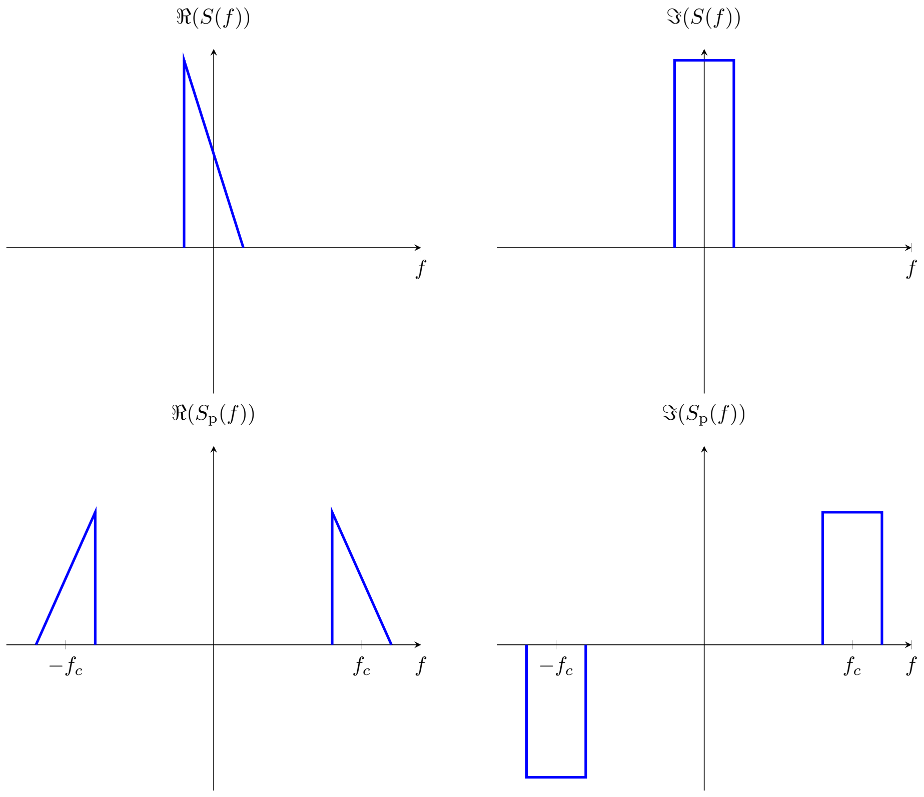


Figure 4: Spectrum of complex envelope and passband signals

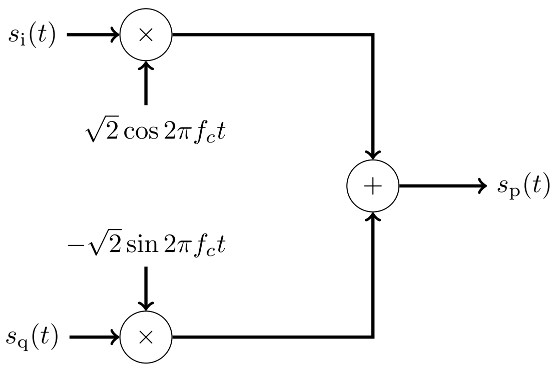


Figure 5: Up-conversion: Getting passband signal from complex envelope

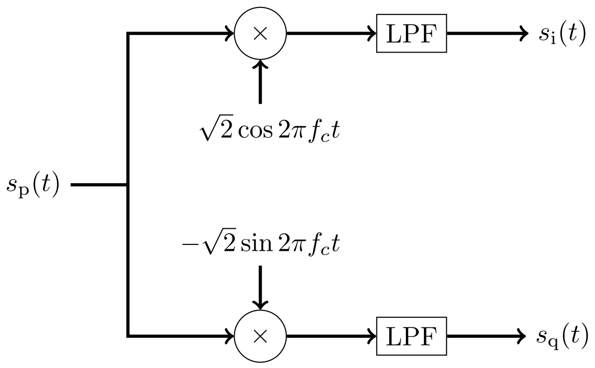


Figure 6: Down-conversion: Getting the complex envelope from passband

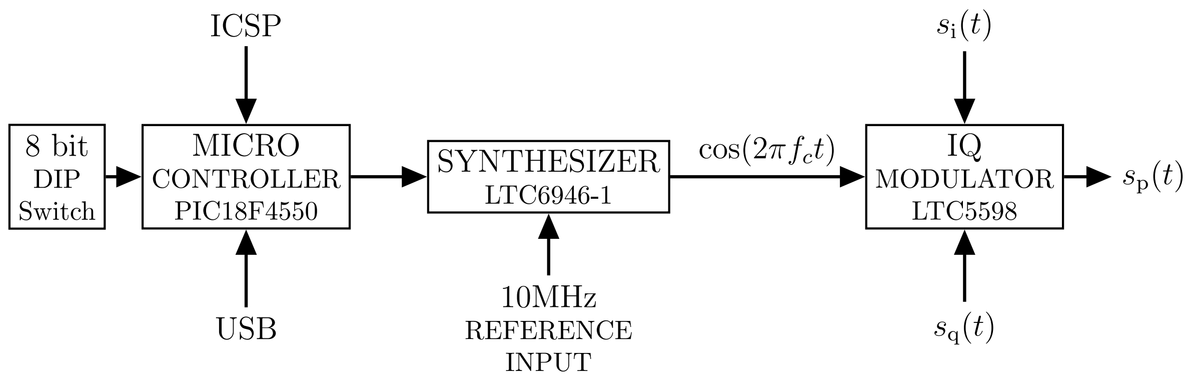


Figure 7: Block diagram of IQ modulator board

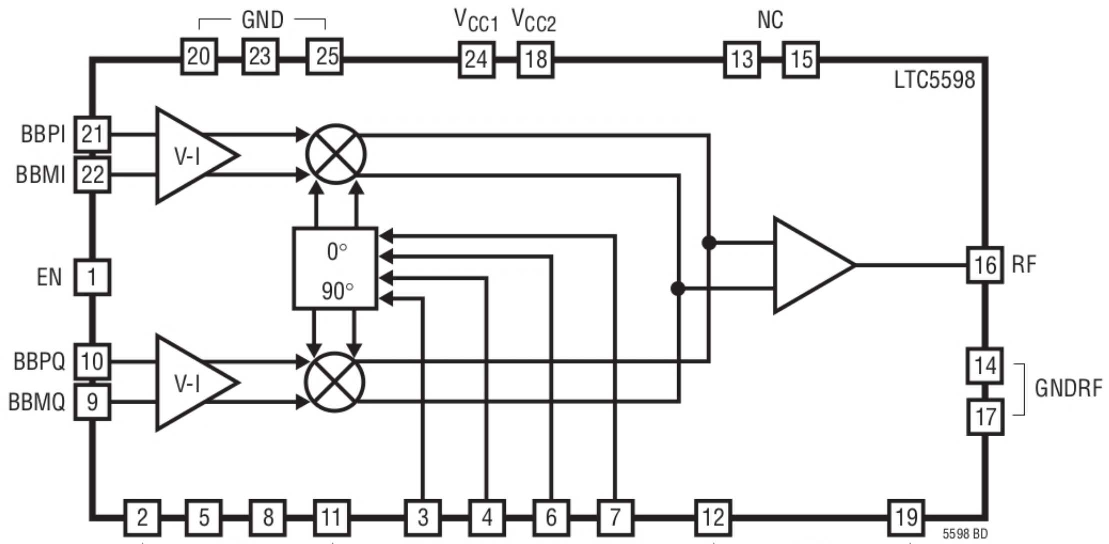


Figure 8: Internal block digram of LTC5598 chip.

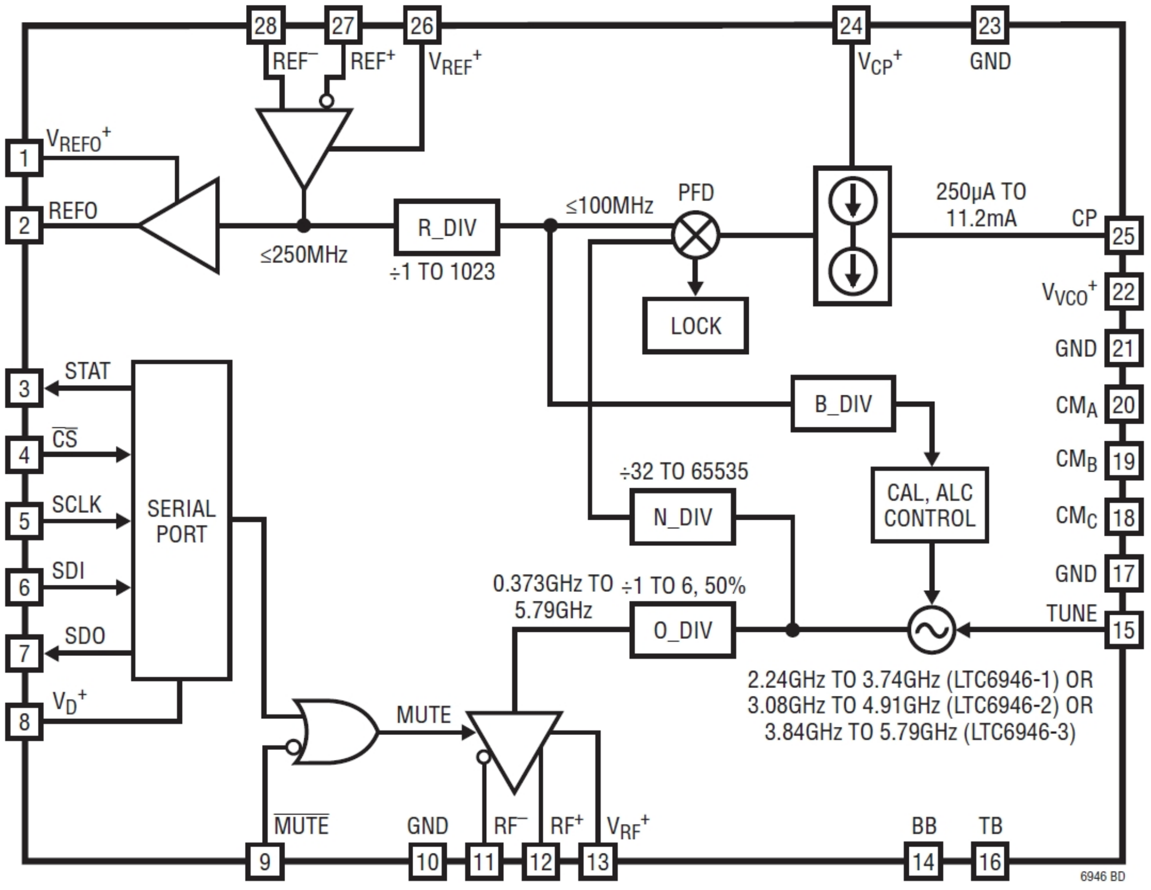


Figure 9: Internal block digram of LTC6946 chip.

a DC voltage(0.5V fixed/0-0.6V adjustable depending on jumper setting) on the board. The LO path consists of an LO buffer with single-ended or differential inputs, and precision quadrature generators that produce the LO drive for the mixers. In our design, the carrier at f_c(LO) generated from LTC6946-1 is fed through the LOP and LOM differential inputs. The passband output $s_{\mathrm{p}}(t)$ can be fed to an antenna from the RF(pin 16) output.

LTC6946-1[3] is a high performance, low noise phase-locked loop (PLL) with a fully integrated VCO, including a reference divider, phase-frequency detector (PFD) with phase-lock indicator, ultralow noise charge pump, integer feedback divider, and VCO output divider. It's frequency range is from 373MHz to 3740MHz. The RF$^-$ and the RF$^+$ outputs from the PLL are respectively connected to the LOM and LOP input of the modulator. The internal block diagram of LTC6946-1 is given in Figure 9.

In our design the reference input of the PLL is fed from a 10MHz crystal oscillator at the REF$^+$(pin 27) input. There are 12 registers in the PLL. We name them h00 to h0B. These registers can be written of read through the chip's SPI compatible serial port. We use PIC18F4550 as the SPI controller. The output frequency is controlled by values in the register. A 10-bit divider, R_DIV, is used to reduce the frequency seen at the PFD. Its divide ratio R may be set to any integer from 1 to 1023, inclusive. The 16-bit N-divider provides the feedback from the VCO to the PFD. Its divide ratio N may be set to any integer from 32 to 65535, inclusive. The 3-bit O-divider can reduce the frequency from the VCO to extend the output frequency range. Its divide ratio O may be set to any integer from 1 to 6, inclusive, outputting a 50% duty cycle even with odd divide values. The frequency at RF$^\pm$ is given by

$$f_{\mathrm{RF}} = \frac{f_{\mathrm{REF}} * N}{O * R} \tag{3}$$

The PIC18F4550 which is the SPI master, loads the registers in the PLL with default configuration at the system startup. We have defined the following default configuration values among others.

$$R = 4$$
$$O = 5$$

With these settings, when the N value changes by 1, the f_{RF} changes by f_{STEP}, where

$$\begin{aligned} f_{\text{STEP}} &= \frac{f_{\text{REF}}}{O * R} \qquad (4) \\ &= \frac{10}{4 * 5}\text{MHz} \\ &= 500\text{kHz} \end{aligned}$$

The N value can be changed using the 8 bit DIP switch. The switch setting $S \in \{0, 1, \cdots, 255\}$ is read by the micro-controller. The default program defines the base value N_0 of N to be 0x0380(890). The value of N is given by

$$N = N_0 + S \qquad (5)$$

Thus with the default settings we can generate f_c, where

$$\begin{aligned} f_c &= f_{\text{STEP}} * (890 + S)\text{kHz}, \quad S \in \{0, 1, \cdots, 255\} \\ &= (448 + 0.5S)\text{MHz}, \quad S \in \{0, 1, \cdots, 255\} \qquad (6) \end{aligned}$$

One can modify the default settings in the PLL by programming the micro-controller, and can tune to other frequencies. The details of how to program the PLL is given in [3]. We have provided ICSP interface and USB interface for programming the micro-controller.

4 RTL-SDR Dongles[5]

Realtek RTL2832U DVB-T tuner dongles are used for DAB/DAB+/FM demodulation. Since the RTL2832U chip allows transferring the raw I/Q samples to the host, it can be used as a cheap SDR. The RTL2832U outputs 8-bit I/Q-samples, and the highest theoretically possible sample-rate is 3.2 MS/s. So theoretically it can capture a signals of upto 3.2MHz passband bandwidth. We usually work at lower sampling rates, so that samples are not lost. The frequency range of operation is determined by the tuner in the circuit. The dongles in our lab have Rafael Micro tuner, whose frequency range of operation is 24 - 1766 MHz.

In short these dongles do what is shown in Figure 6 and gives the samples of $s_{\text{i}}(t)$ and $s_{\text{q}}(t)$ to the host computer. A GNU Radio flow-graph in the host computer can read these samples and these samples can be processed using various DSP algorithms in GNU Radio, to extract the embedded information.

5 Universal Software Radio Peripheral(USRP)[6]

USRP is a software radio platform, which usually have both transmit and receive capabilities. USRP can be directly interfaced with a computer through USB of Gigabit Ethernet. When used as a transmitter a USRP essentially do what is shown in Figure 5, in addition to generation of both of both the $s_{\text{i}}(t)$ and $s_{\text{q}}(t)$, from their respective samples supplied by the host computer, through digital to analog conversion. When used a receiver, a USRP essentially do what is shown in Figure 6 and gives the samples of $s_{\text{i}}(t)$ and $s_{\text{q}}(t)$ to the host computer.

In our lab we will be using USRP B100 series. B100 USRPs have USB 2.0 interface. We can set the sample width to 8-bits or 16-bits. With 8 bit sample width, each of I and Q sample takes 8 bits each, and thus a complex sample in total takes 16 bits. With 16 bit sample width, each of I and Q sample takes 16 bits each, and thus a complex sample in total takes 32 bits. The speed of of USB 2.0 interface is 480Mbps. So we can get maximum sampling rate of 30MS/s at 8 bit sample width. Thus we can capture/send messages of bandwidth upto 30MHz. With 16 bit sample width we can get maximum sampling rate of 15MS/s, limiting our maximum bandwidth to 15MHz.

The frequency range of operation of the USRP is determined by the RF daughter board attached. In the labs we will be using WBX daughter boards which can operate in 50 MHz to 2.2 GHz range.

References

[1] http://gnuradio.org

[2] http://www.linear.com/product/LTC5598

[3] http://www.linear.com/product/LTC6946

[4] Upamanyu Madhov, “Fundamentals of Digital Communication”, CUP 2008.

[5] http://sdr.osmocom.org/trac/wiki/rtl-sdr

[6] http://www.ettus.com/home

Appendices

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Audioplay1
# Generated: Sat Jun 28 18:50:14 2014
##################################################

from gnuradio import analog
from gnuradio import audio
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class audioplay1(grc_wxgui.top_block_gui):

    def __init__(self):
        grc_wxgui.top_block_gui.__init__(self, title="Audioplay1")
        _icon_path = "/usr/share/icons/hicolor/32x32/apps/gnuradio-grc.png"
        self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))

        ##################################################
        # Variables
        ##################################################
        self.samp_rate = samp_rate = 32000

        ##################################################
        # Blocks
        ##################################################
        self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vff((0.5, ))
        self.audio_sink_0 = audio.sink(samp_rate, "", True)
        self.analog_sig_source_x_0 = analog.sig_source_f(samp_rate, analog.GR_COS_WAVE,
    500, 0.2, 0)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.audio_sink_0, 0))
        self.connect((self.analog_sig_source_x_0, 0), (self.blocks_multiply_const_vxx_0,
     0))



    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.analog_sig_source_x_0.set_sampling_freq(self.samp_rate)

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    (options, args) = parser.parse_args()
    tb = audioplay1()
    tb.Start(True)
    tb.Wait()
```

Listing 1: The python program corresponding to the flow-graph in Figure 1